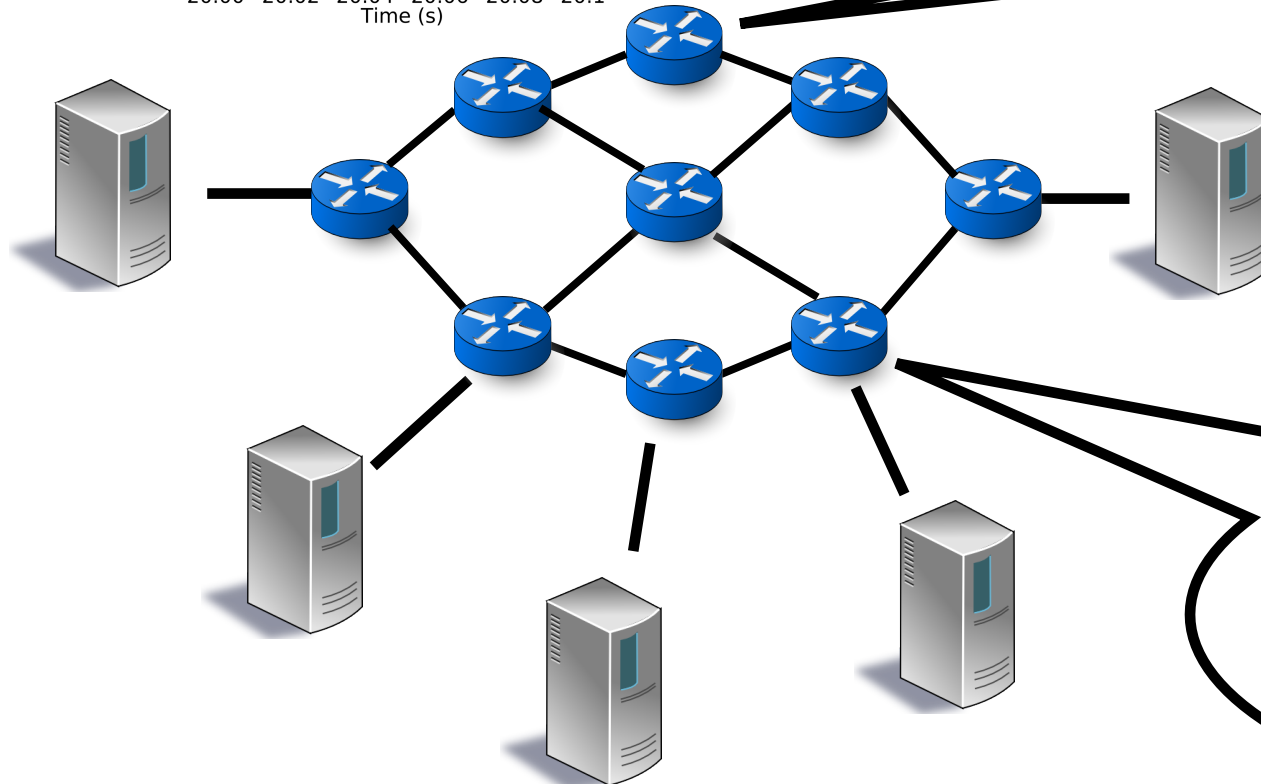
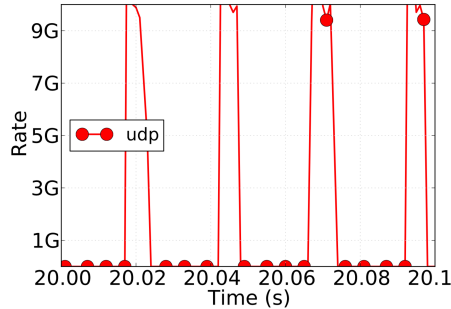


# Language-Directed Hardware Design for Network Performance Monitoring

**Srinivas Narayana**, Anirudh Sivaraman, Vikram Nathan,  
Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimal  
Jeyakumar, and Changhoon Kim



# Example: Who caused a microburst?



Queue build-up deep in the network

- ✗ End-to-end probes
- ✗ Sampling
- ✗ Counters & Sketches
- ? Mirror packets

Per-pkt info: challenging in software  
6.4Tbit/s switch: Need **100M** recs/s  
COTS: **100K-1M** recs/s/core

Switches should be first-class citizens in performance monitoring.

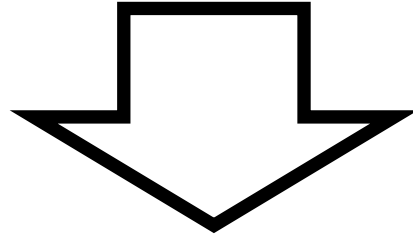
# Why monitor from switches?

- Already see the queues & concurrent connections
- Infeasible to stream all the data out for external processing
- Can we filter and aggregate performance on switches directly?

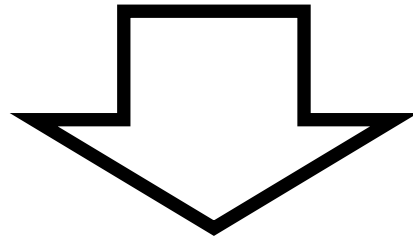
We want to build “future-proof” hardware:

Language-directed hardware design

# Performance monitoring use cases



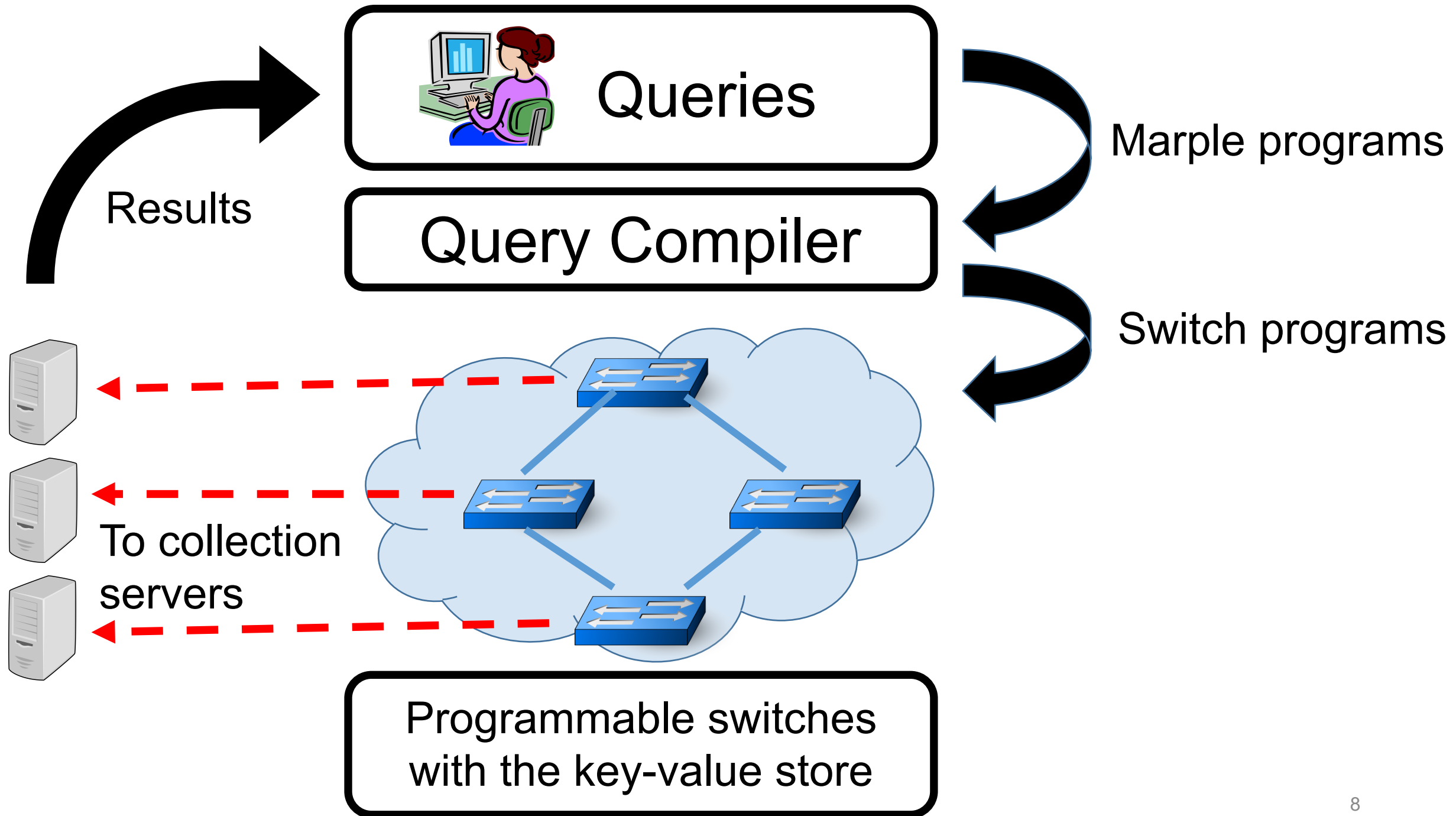
## Expressive query language



## Line-rate switch hardware primitives

# Contributions

- **Marple**, a performance query language
- Line-rate switch hardware design
  - Aggregation: **Programmable key-value store**
- Query compiler



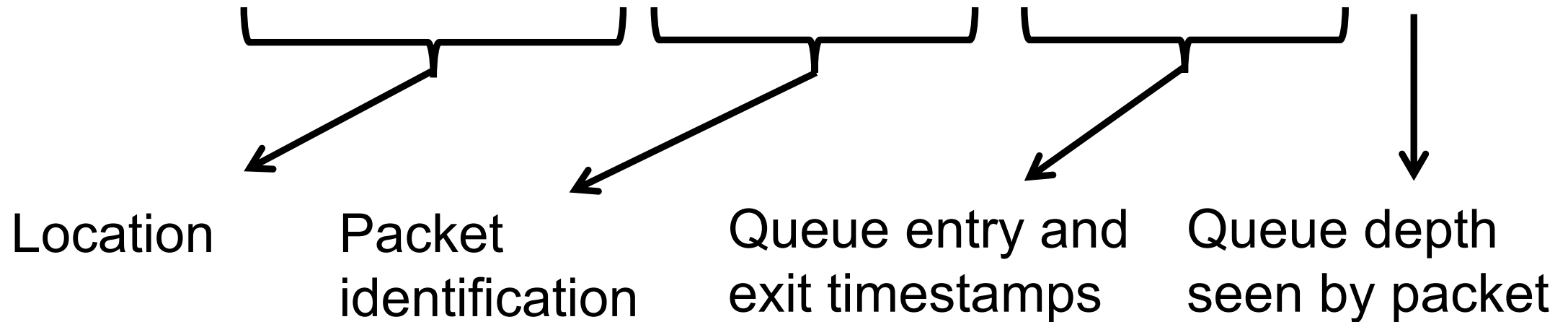


# Marple: Performance query language

# Marple: Performance query language

Stream: For *each* packet at *each* queue,

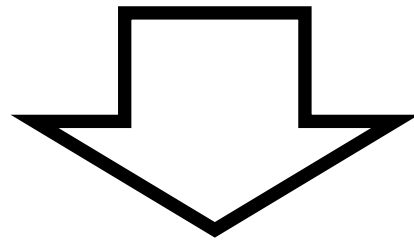
$S := (\text{switch}, \text{qid}, \text{hdrs}, \text{uid}, \text{tin}, \text{tout}, \text{qsize})$



# Marple: Performance query language

Stream: For *each* packet at *each* queue,

$S := (\text{switch}, \text{qid}, \text{hdrs}, \text{uid}, \text{tin}, \text{tout}, \text{qsize})$



Familiar functional operators

filter

map

zip

groupby

# Example: High queue latency packets

```
R1 = filter(S, tout - tin > 1 ms)
```

# Example: Per-flow average latency

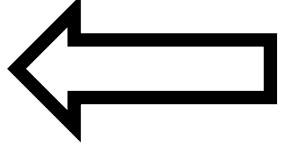
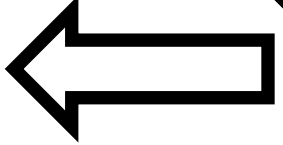
```
R1 = filter(S, proto == TCP)
R2 = groupby(R1, 5tuple, ewma)
```



Fold function

```
def ewma([avg], [tin, tout]):
    avg = (1- $\alpha$ )*avg +  $\alpha$ *(tout-tin)
```

# Example: Microburst diagnosis

```
def bursty([last_time, nbursts], [tin]):  
    if tin - last_time > 800 ms:   
        nbursts = nbursts + 1   
    last_time = tin
```

```
result = groupby(S, 5tuple, bursty)
```

# Many performance queries (see paper)

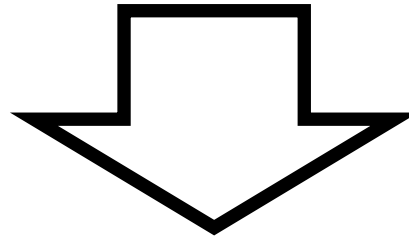
- Transport protocol diagnoses
  - Fan-in problems (incast and outcast)
  - Incidents of reordering and retransmissions
  - Interference from bursty traffic
- Flow-level metrics
  - Packet drop rates
  - Queue latency EWMA per connection
  - Incidence and lengths of flowlets
- Network-wide questions
  - Route flapping
  - High end to end latencies
  - Locations of persistently long queues
- ...

# Implementing Marple on switches



# Implementing Marple on switches

$S := (\text{switch}, \text{hdrs}, \text{uid}, \text{qid}, \text{tin}, \text{tout}, \text{qsize})$



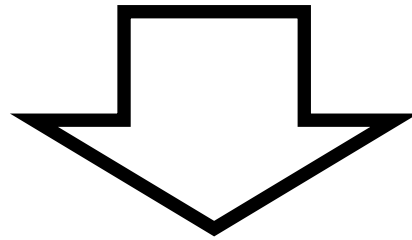
Switch telemetry  
[INT SOSR'15]



Stateless match-action rules  
[RMT SIGCOMM'13]

# Implementing Marple on switches

$S := (\text{switch}, \text{hdrs}, \text{uid}, \text{qid}, \text{tin}, \text{tout}, \text{qsize})$



Switch telemetry  
[INT SOSR'15]



Stateless match-action rules  
[RMT SIGCOMM'13]

# Implementing aggregation

```
ewma_query = groupby(S, 5tuple, ewma)
def ewma([avg], [tin, tout]):
    avg = (1- $\alpha$ )*avg +  $\alpha$ *(tout-tin)
```

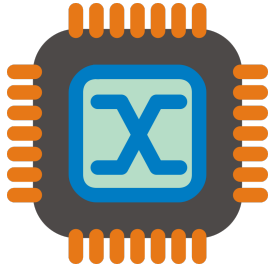
- Compute & update values at switch line rate (1 pkt/ns)
- Scale to millions of aggregation keys (e.g., 5-tuples)

Challenge:

Neither SRAM nor DRAM is both **fast** and **dense**

Caching:  
the illusion of fast and large memory

# Caching



On-chip cache  
(SRAM)

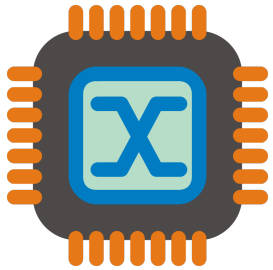
Key	Value



Off-chip backing  
store (DRAM)

Key	Value

# Caching



On-chip cache  
(SRAM)

Key	Value
✓	

*Read* value for  
5-tuple key K

→  
*Modify* value  
using ewma

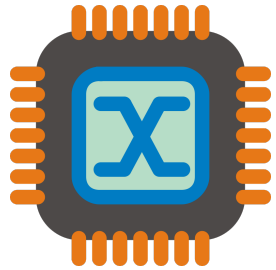
*Write back*  
updated value



Off-chip backing  
store (DRAM)

Key	Value

# Caching



*Read value for  
5-tuple key K*

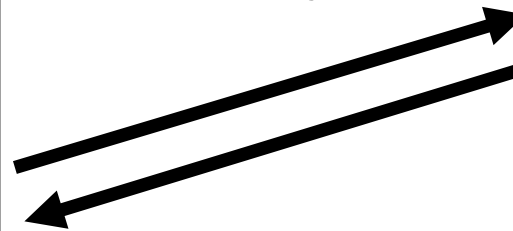


On-chip cache  
(SRAM)

Key	Value



Req. key K



Resp.  $V_{back}$

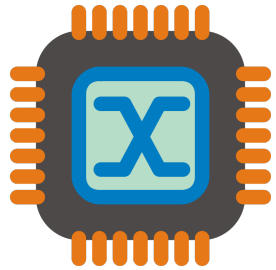


Off-chip backing  
store (DRAM)

Key	Value
K	$V_{back}$



# Caching



*Read value for  
5-tuple key K*



On-chip cache  
(SRAM)

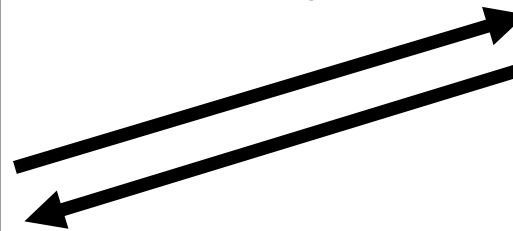
Key	Value
K	$V_{\text{back}}$



Off-chip backing  
store (DRAM)

Key	Value
K	$V_{\text{back}}$

Req. key K



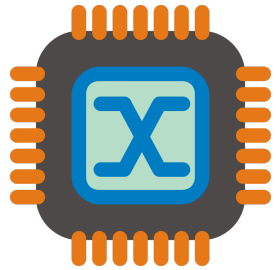
Resp.  $V_{\text{back}}$

Modify and write must wait for DRAM.

*Non-deterministic latencies stall packet pipeline.*

Instead, we treat cache misses as packets from new flows.

# Cache misses as new keys



Read value for  
key K



On-chip cache  
(SRAM)

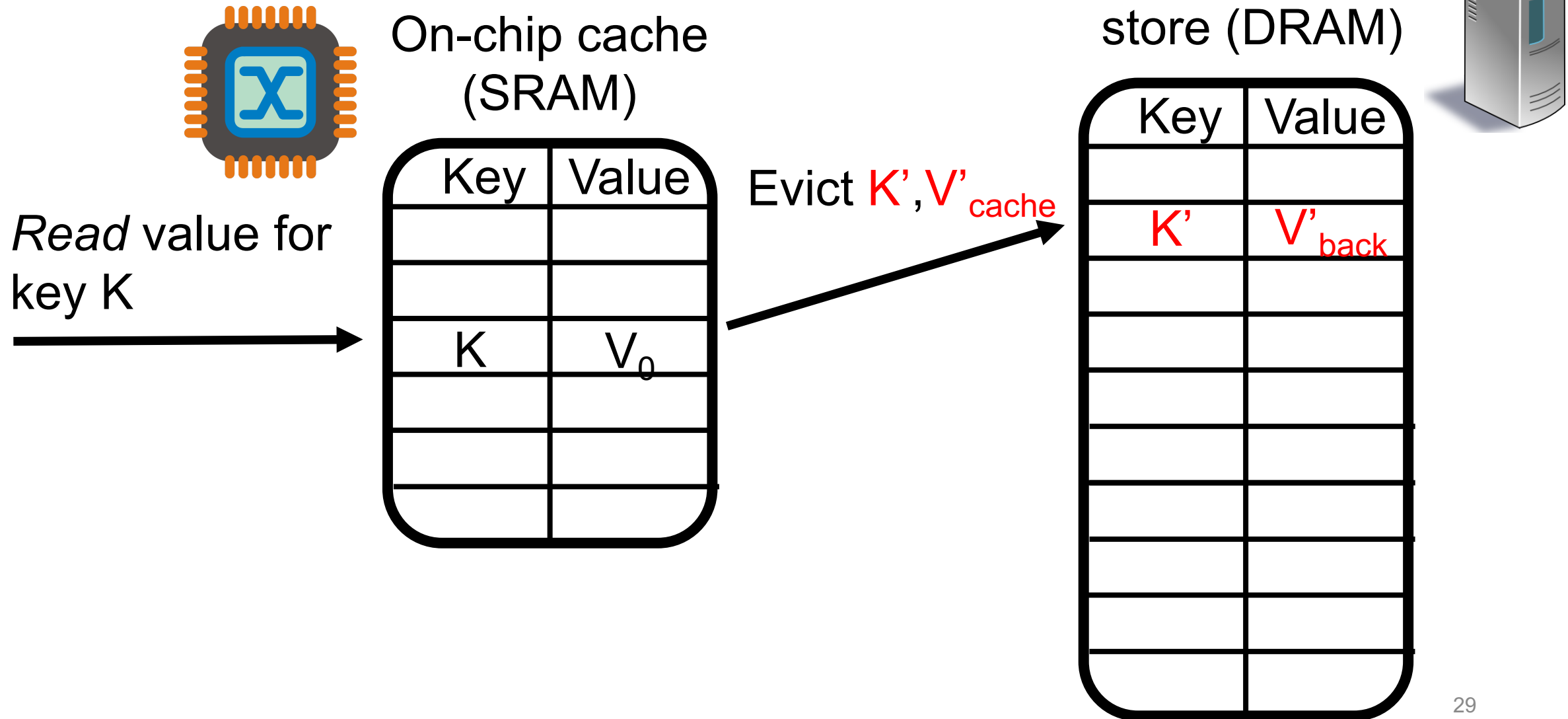
Key	Value
K <del>X</del>	$V_0$

Off-chip backing  
store (DRAM)

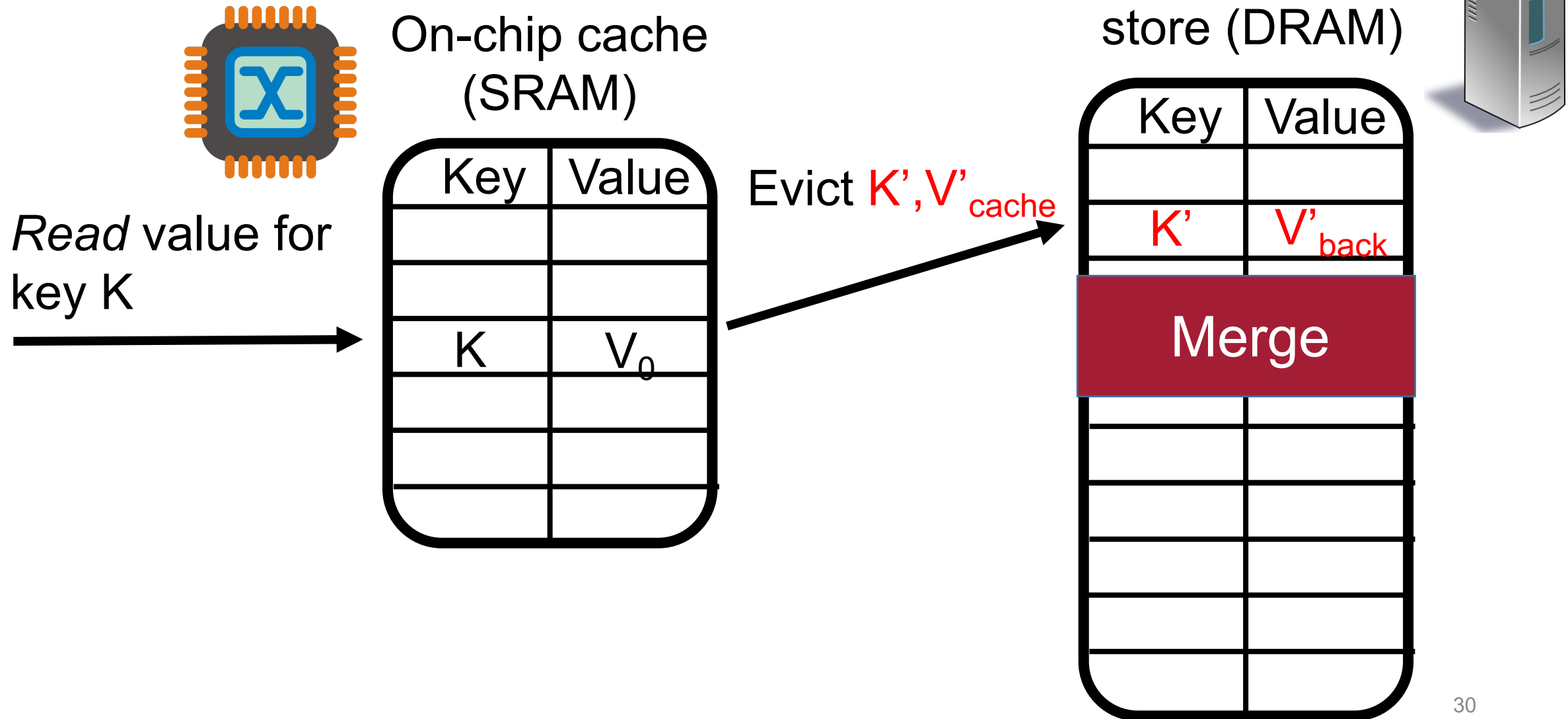


Key	Value

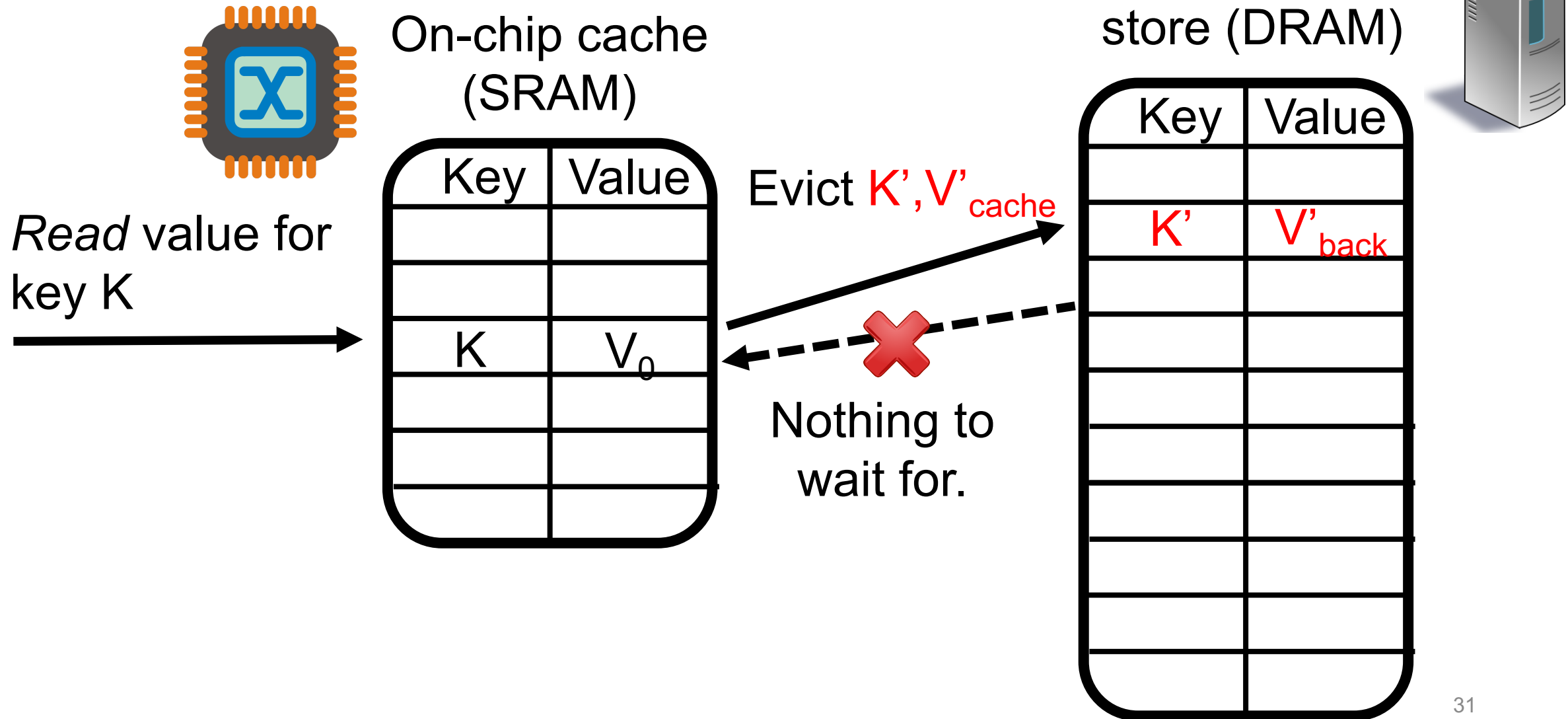
# Cache misses as new keys



# Cache misses as new keys



# Cache misses as new keys



# Cache misses as new keys

On-chip cache

Off-chip backing  
store (DRAM)

Packet processing doesn't wait for DRAM.

Retain 1 pkt/ns processing rate!👍





# How about value accuracy after evictions?

- Merge should “accumulate” results of folds across evictions
- Suppose: Fold function  $g$  over a packet sequence  $p_1, p_2, \dots$

$$g([p_i])$$



Action of  $g$  over a packet sequence, e.g., EWMA

# The Merge operation

$$\begin{aligned} & \text{merge}(\overbrace{g([q_j])}^{V_{\text{cache}}}, \overbrace{g([p_i])}^{V_{\text{back}}}) \\ &= \underbrace{g([p_1, \dots, p_n, q_1, \dots, q_m])}_{\text{Fold over the entire packet sequence}} \end{aligned}$$

- Example: if  $g$  is a counter, merge is just addition!

# Mergeability

- Can merge any fold  $g$  by storing entire pkt sequence in cache
  - ... but that's a lot of extra state!
- Can we merge a given fold with “small” extra state?
- Small: extra state size  $\approx$  size of the state used by the fold itself

There are useful fold functions that require a large amount of extra state to merge.

(see formal result in paper)

# Linear-in-state: Mergeable w. small extra state

$$S = A * S + B$$

State of the fold function

Functions of a bounded number of packets in the past

- Examples: Packet and byte counters, EWMA, functions over a window of packets, ...

# Example: EWMA merge operation

- EWMA :  $S = (1-\alpha)*S + \alpha*(\text{func of current packet})$

$$\text{merge}(V_{\text{cache}}, V_{\text{back}}) = V_{\text{cache}} + (1-\alpha)^N * (V_{\text{back}} - V_0)$$

Small extra state

N: # pkts processed by cache

# Microbursts: Linear-in-state!

```
def bursty([last_time, nbursts], [tin]):  
    if tin - last_time > 800 ms:  
        nbursts = nbursts + 1  
    last_time = tin
```

```
result = groupby(S, 5tuple, bursty)
```

nbursts:  $S = A * S + B$ , where

$A = 1$

$B = \begin{cases} 1, & \text{if current pkt within time gap from last;} \\ 0 & \text{otherwise} \end{cases}$

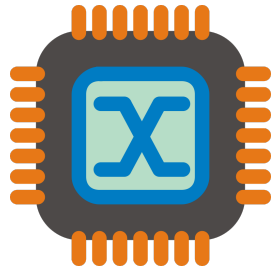
# Other linear-in-state queries

- Counting successive TCP packets that are out of order
- Histogram of flowlet sizes
- Counting number of timeouts in a TCP connection
- ... 7/10 example queries in our paper



Evaluation:  
Is processing the evictions feasible?

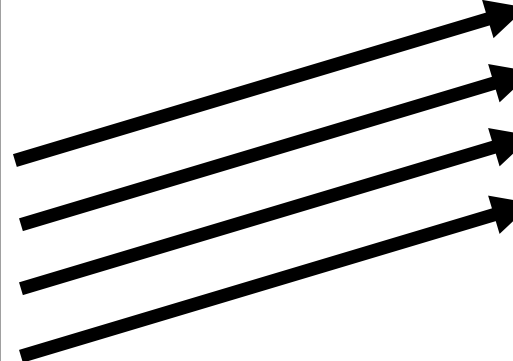
# Eviction processing



On-chip cache  
(SRAM)

Key	Value
K	$V_0$

Evict  $K', V'$  <sub>cache</sub>



Off-chip backing  
store (DRAM)

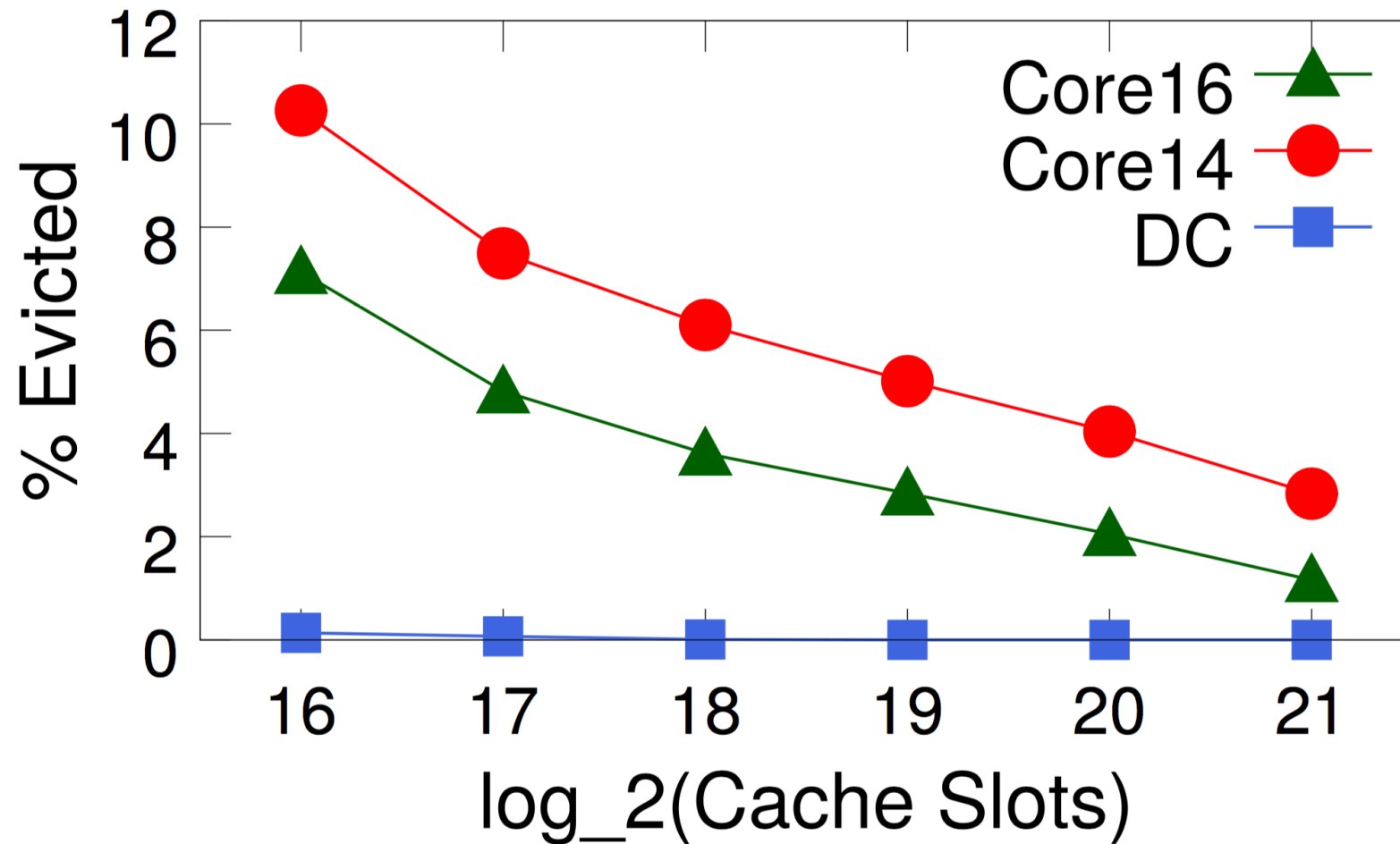


Key	Value
$K'$	$V'_{\text{back}}$

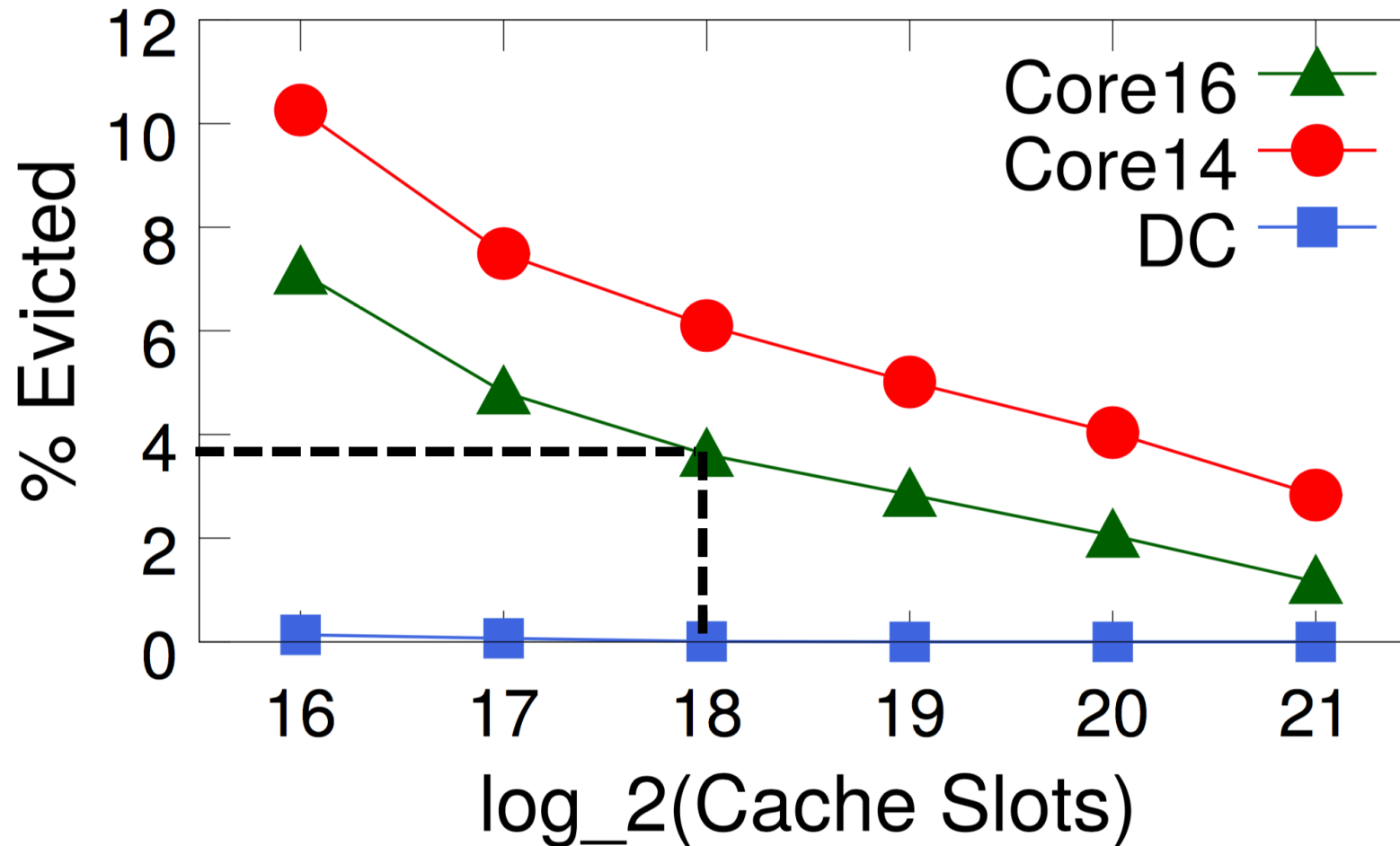
# Eviction processing at backing store

- Trace-based evaluation:
  - “Core14”, “Core16”: Core router traces from CAIDA (2014, 16)
  - “DC”: University data center trace from [Benson et al. IMC '10]
  - Each has ~100M packets
- Query aggregates by 5-tuple (key)
  - Show results for key+value size of 256 bits
- 8-way set-associative LRU cache eviction policy
- Eviction *ratio*: % of incoming pkts that result in a cache eviction<sub>43</sub>

# Eviction ratio vs. Cache size



# Eviction ratio vs. Cache size



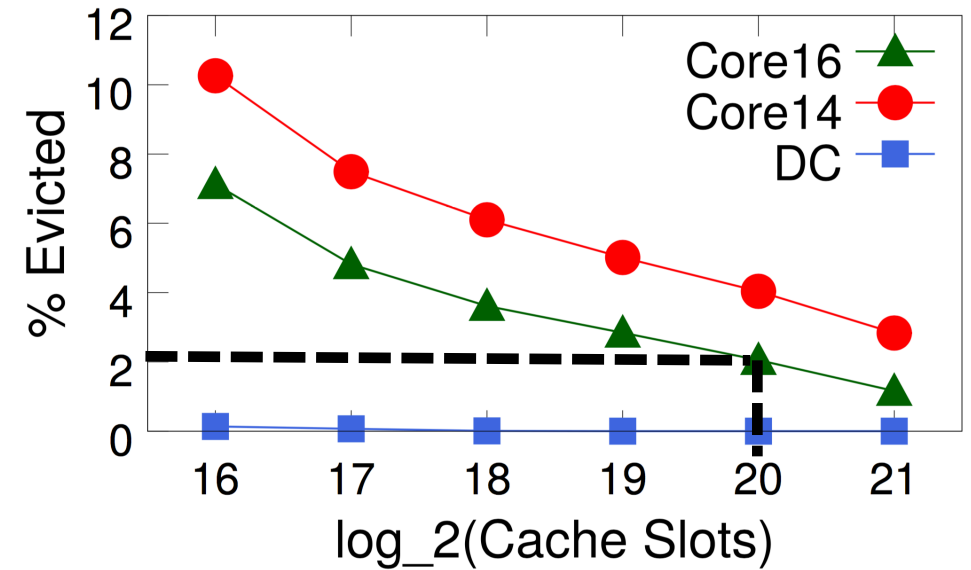
$2^{18}$  keys == 64 Mbits

4% pkt eviction ratio

**25X** reduction from  
processing each pkt

# Eviction ratio → Eviction rate

- Consider 64-port X 100-Gbit/s switch
- Memory: 256 Mbits
  - 7.5% area
- Eviction rate: 8M records/s
  - ~ **32 cores**



# See more in the paper...

- More performance query examples
- Query compilation algorithms
- Evaluating hardware resources for stateful computations
- Implementation & end-to-end walkthroughs on mininet

# Summary

Come see our demo!

- On-switch aggregation reduces software data processing
- Linear in state: fully accurate per-flow aggregation at line rate

alephtwo@csail.mit.edu

<http://web.mit.edu/marple>

